PHASE1-1.3 PART 1: GCP Cost Collector - Code Generation

OptiInfra Development Series
Phase: Cost Agent (Week 2-3)
Component: GCP Cost Metrics Collection
Estimated Time: 20 minutes setup + 15 minutes validation
Dependencies: P-01 (Bootstrap), 1.1 (Cost Agent Skeleton), 1.2 (AWS Collector Pattern), 0.10 (Shared Utilities)

📋 Overview

This prompt creates a complete GCP cost collection system that integrates with Google Cloud Billing API to gather Compute Engine, Cloud SQL, Cloud Functions, Cloud Storage, and other service costs. It identifies optimization opportunities and stores metrics in ClickHouse for analysis.

🎯 Objectives

By the end of this prompt, you will have:

1. ✅ Google Cloud Billing API integration
2. ✅ GCE, Cloud SQL, Cloud Functions, GCS cost collectors
3. ✅ Automated cost analysis and anomaly detection
4. ✅ Idle resource identification for GCP
5. ✅ Cost metrics stored in ClickHouse
6. ✅ Comprehensive unit and integration tests
7. ✅ Prometheus metrics for GCP cost collection

🔧 Prerequisites

Before starting, ensure:

- ☐ PHASE1-1.2 completed - AWS Collector pattern established
- ☐ GCP project with Billing API enabled
- ☐ GCP service account credentials (JSON key file)
- ☐ google-cloud-billing library installed
- ☐ ClickHouse running with cost metrics tables
- ☐ At least 2 weeks of GCP usage data available

📝 Detailed Windsurf Prompt

Create a complete GCP cost collection system for OptiInfra Cost Agent, following the pattern established in the AWS collector (1.2).

CONTEXT:
- Multi-agent LLM infrastructure optimization platform
- Cost Agent already has skeleton + AWS collector
- Need to collect real GCP cost data via Cloud Billing API
- Analyze Compute Engine, Cloud SQL, Cloud Functions, Cloud Storage costs
- Identify idle resources and optimization opportunities
- Store metrics in ClickHouse for time-series analysis
- Project root: ~/optiinfra/services/cost-agent

REQUIREMENTS:

1. GCP COST COLLECTOR BASE (src/collectors/gcp/)
   Create base GCP collector class (similar to AWS base.py):
   - __init__.py - Module initialization
   - base.py - Base GCP collector with common functionality
     * GCP client initialization with service account
     * Credential handling (JSON key file, ADC)
     * Project selection and multi-project support
     * Retry logic with exponential backoff
     * Error handling and logging
     * Connection pooling
     * Rate limit handling (Billing API: 300 req/min per project)

   Base class should include:
   - get_credentials() - Load service account credentials
   - get_client(service_name) - Returns Google client
   - paginate_results() - Helper for paginated API calls
   - handle_rate_limiting() - Automatic retry with backoff
   - log_api_call() - Track API usage for rate limiting
   - get_all_projects() - List accessible GCP projects

2. BILLING API CLIENT (src/collectors/gcp/billing_client.py)
   Create Cloud Billing wrapper:
   - BillingClient class
   - get_billing_info() - Retrieve billing account details
   - query_costs() - BigQuery-based cost retrieval
     * Use Cloud Billing export to BigQuery
     * Date range: configurable (default: last 30 days)

* Group by: service, SKU, project, region
* Support for credits, discounts, taxes
- get_cost_forecast() - 30-day cost prediction
- get_committed_use_discounts() - CUD utilization
- get_sustained_use_discounts() - SUD calculations
- Data transformation to OptiInfra format

BigQuery SQL template:
```sql
SELECT
  DATE(usage_start_time) as date,
  service.description as service,
  location.region as region,
  project.id as project_id,
  SUM(cost) as cost,
  SUM(usage.amount) as usage_amount,
  usage.unit as usage_unit
FROM `{project}.{dataset}.gcp_billing_export_v1_{billing_account_id}`
WHERE DATE(usage_start_time) BETWEEN @start_date AND @end_date
GROUP BY date, service, region, project_id, usage_unit
ORDER BY date DESC, cost DESC
```

Response format:
{
  "time_period": {"start": "2025-10-01", "end": "2025-10-31"},
  "total_cost": 95000.75,
  "by_service": {
    "Compute Engine": 65000.00,
    "Cloud SQL": 18000.00,
    "Cloud Functions": 7000.75
  },
  "by_project": {...},
  "by_region": {...}
}

3. COMPUTE ENGINE COLLECTOR (src/collectors/gcp/compute_engine.py)
Create GCE-specific collector:
- ComputeEngineCostCollector class
- collect_instance_costs() - Per-instance cost breakdown
- identify_idle_instances() - CPU <5%, Network <1GB/day

- identify_underutilized_instances() - CPU <20% for 14+ days
- get_preemptible_opportunities() - Instances eligible for preemptible
- get_committed_use_recommendations() - CUD purchase suggestions
- analyze_instance_types() - Usage patterns by machine type
- get_persistent_disk_costs() - Disk costs (attached + unattached)
- identify_unattached_disks() - Disks not attached to instances
- analyze_snapshot_costs() - Snapshot storage costs

Use Cloud Monitoring for utilization metrics:
- CPU utilization (14-day average)
- Network sent/received (14-day average)
- Disk read/write ops

Output format:

```
{
  "instance_id": "1234567890123456789",
  "instance_name": "instance-1",
  "machine_type": "n2-standard-4",
  "zone": "us-central1-a",
  "monthly_cost": 180.00,
  "utilization": {
    "cpu_avg": 12.5,
    "network_gb_day": 350.0
  },
  "optimization": {
    "is_idle": false,
    "is_underutilized": true,
    "preemptible_eligible": true,
    "rightsizing_recommendation": "n2-standard-2",
    "estimated_savings": 90.00
  }
}
```

4. CLOUD SQL COLLECTOR (src/collectors/gcp/cloud_sql.py)
Create Cloud SQL-specific collector:
- CloudSQLCostCollector class
- collect_sql_costs() - Per-instance costs
- identify_idle_databases() - Connection count = 0
- analyze_storage_costs() - Storage and backup costs
- get_committed_use_recommendations() - Cloud SQL CUDs
- identify_high_availability_opportunities() - Convert to zonal

- analyze_backup_costs() - Backup retention optimization
- check_auto_scaling() - Recommendations for auto-scaling

Cloud Monitoring metrics:
- database/cpu/utilization (14-day average)
- database/disk/utilization
- database/memory/utilization
- database/network/connections

5. CLOUD FUNCTIONS COLLECTOR (src/collectors/gcp/cloud_functions.py)
Create Cloud Functions-specific collector:
- CloudFunctionsCostCollector class
- collect_function_costs() - Per-function costs
- analyze_invocations() - Invocation count and duration
- identify_over_provisioned() - Memory > needed
- calculate_optimal_memory() - Cost-performance optimization
- identify_cold_starts() - Functions with high cold start %
- analyze_timeout_issues() - Functions frequently timing out

Cloud Monitoring metrics:
- function/execution_count
- function/execution_times
- function/user_memory_bytes
- function/active_instances

6. CLOUD STORAGE COLLECTOR (src/collectors/gcp/cloud_storage.py)
Create GCS-specific collector:
- CloudStorageCostCollector class
- collect_bucket_costs() - Per-bucket costs
- analyze_storage_classes() - Standard vs Nearline vs Coldline vs Archive
- identify_lifecycle_opportunities() - Object lifecycle policies
- calculate_transfer_costs() - Egress charges
- identify_incomplete_uploads() - Cleanup opportunities
- analyze_bucket_usage() - Access patterns

7. COST ANALYZER (src/analyzers/gcp_analyzer.py)
Create comprehensive analyzer:
- GCPCostAnalyzer class
- analyze_all_services() - Aggregate all collectors
- detect_anomalies() - Unusual cost spikes (>20% change)
- calculate_waste() - Total waste across all services

- prioritize_opportunities() - Sort by savings potential
- compare_with_aws() - Multi-cloud cost comparison
- generate_summary_report() - Executive summary

Analysis output:

```
{
  "total_monthly_cost": 95000.75,
  "total_waste": 42000.00,
  "waste_percentage": 44.2,
  "opportunities": [
    {
      "type": "preemptible_migration",
      "service": "Compute Engine",
      "instance_count": 12,
      "estimated_savings": 15600.00,
      "confidence": 0.82,
      "priority": "high"
    },
    {
      "type": "idle_resource",
      "service": "Cloud SQL",
      "resource_count": 2,
      "estimated_savings": 9000.00,
      "confidence": 0.93,
      "priority": "high"
    }
  ],
  "anomalies": [
    {
      "service": "Cloud Functions",
      "metric": "invocation_cost",
      "change_percentage": 38.0,
      "date": "2025-10-15"
    }
  ]
}
```

8. BIGQUERY INTEGRATION (src/collectors/gcp/bigquery_helper.py)

Create BigQuery helper for cost queries:
- BigQueryHelper class
- execute_billing_query() - Run cost queries

- get_daily_costs() - Daily breakdown
- get_service_costs() - Costs by service
- get_project_costs() - Costs by project
- get_sku_details() - Detailed SKU-level costs
- cache_query_results() - Cache in Redis
- export_to_clickhouse() - Store in ClickHouse

Cost query examples:
- Most expensive services
- Cost trends over time
- Unexpected cost spikes
- Credits and discounts applied
- Project-level breakdown

9. CLICKHOUSE STORAGE (src/storage/gcp_metrics.py)
Create ClickHouse integration:
- GCPMetricsStorage class
- store_cost_metrics() - Daily cost time-series
- store_instance_metrics() - Per-resource metrics
- store_optimization_opportunities() - Discovered opportunities
- query_cost_trends() - Retrieve historical data
- query_by_service() - Filter by service
- query_by_project() - Filter by project
- query_multi_cloud() - Compare AWS vs GCP costs

Use existing ClickHouse tables from 0.3:
- cost_metrics (timestamp, provider, service, project, cost)
- resource_metrics (timestamp, provider, resource_id, utilization)
- optimization_opportunities (timestamp, provider, type, savings)

10. API ENDPOINTS (src/api/gcp_costs.py)
Create FastAPI endpoints:
- POST /api/v1/gcp/collect - Trigger collection
- GET /api/v1/gcp/costs - Retrieve cost data
  * Query params: start_date, end_date, service, project, region
- GET /api/v1/gcp/opportunities - Get optimization opportunities
  * Query params: min_savings, service, priority
- GET /api/v1/gcp/analysis - Get comprehensive analysis
- POST /api/v1/gcp/refresh - Force refresh from GCP
- GET /api/v1/gcp/compare-aws - Multi-cloud comparison

All endpoints should:
- Follow same pattern as AWS endpoints
- Validate input with Pydantic models
- Handle authentication/authorization
- Return standardized responses
- Log all requests
- Update Prometheus metrics

11. PROMETHEUS METRICS (update src/metrics.py)
Add GCP-specific metrics (mirror AWS pattern):
- gcp_api_calls_total - Counter by service
- gcp_api_errors_total - Counter by error type
- gcp_cost_collection_duration_seconds - Histogram
- gcp_total_monthly_cost_usd - Gauge by service
- gcp_waste_identified_usd - Gauge by service
- gcp_optimization_opportunities - Gauge by type
- gcp_idle_resources_count - Gauge by service
- gcp_underutilized_resources_count - Gauge by service
- gcp_preemptible_eligible_count - Gauge
- gcp_committed_use_discount_coverage - Gauge

12. CONFIGURATION (update src/config.py)
Add GCP settings:
- GCP_PROJECT_ID - Primary project
- GCP_PROJECTS - List of projects to analyze (optional)
- GCP_CREDENTIALS_PATH - Path to service account JSON
- GCP_BILLING_ACCOUNT_ID - Billing account ID
- GCP_BIGQUERY_DATASET - Dataset with billing export
- GCP_DEFAULT_REGION - Default: us-central1
- GCP_REGIONS - List of regions to analyze
- GCP_COST_LOOKBACK_DAYS - Default: 30
- GCP_IDLE_CPU_THRESHOLD - Default: 5%
- GCP_UNDERUTILIZED_CPU_THRESHOLD - Default: 20%
- GCP_PREEMPTIBLE_SAVINGS_TARGET - Default: 60%
- GCP_COLLECTION_SCHEDULE - Cron expression

13. TESTING (tests/collectors/test_gcp.py)
Create comprehensive tests:
- test_gcp_credentials_loading()
- test_billing_client()
- test_bigquery_integration()

- test_compute_engine_collection()
- test_cloud_sql_collection()
- test_cloud_functions_collection()
- test_cloud_storage_collection()
- test_idle_resource_detection()
- test_preemptible_opportunity_identification()
- test_cost_analyzer()
- test_clickhouse_storage()
- test_api_endpoints()
- test_multi_cloud_comparison()

Use unittest.mock for GCP API mocking:
- Mock BigQuery client and query results
- Mock Compute Engine list instances
- Mock Cloud Monitoring metrics
- Mock Cloud SQL list instances

14. INTEGRATION TESTS (tests/integration/test_gcp_integration.py)
Create E2E tests:
- test_full_collection_workflow()
- test_analysis_pipeline()
- test_storage_retrieval()
- test_api_e2e()
- test_aws_gcp_comparison()

Requirements:
- Use real GCP sandbox project (if available)
- Or comprehensive mocking
- Test error scenarios
- Test rate limiting
- Test multi-project collection

15. DOCUMENTATION (docs/gcp-collector.md)
Create comprehensive docs:
- Setup instructions
- GCP IAM roles required
- BigQuery export setup
- Configuration guide
- API reference
- Example queries
- Troubleshooting guide

- Billing API limits

TECHNICAL SPECIFICATIONS:
- Python 3.11+
- google-cloud-billing >= 1.11.0
- google-cloud-compute >= 1.14.0
- google-cloud-monitoring >= 2.15.0
- google-cloud-bigquery >= 3.11.0
- Pydantic for data validation
- asyncio for concurrent collection
- tenacity for retry logic
- Rate limiting: 300 requests/minute per project
- Batch size: 500 resources per request
- Error handling: Log and continue, don't crash
- Caching: Use Redis for 1-hour cache on BigQuery

BEST PRACTICES:
- Use service account authentication (not user credentials)
- Implement exponential backoff for rate limits
- Cache BigQuery results (expensive queries)
- Parallelize collection across projects (asyncio)
- Use pagination for large result sets
- Validate all GCP API responses
- Log all API calls for debugging
- Use Cloud Monitoring for utilization metrics
- Store raw data + analyzed data separately
- Update metrics after each collection
- Follow AWS collector patterns for consistency

ERROR HANDLING:
- Catch google.api_core.exceptions.GoogleAPIError
- Handle PermissionDenied (IAM issues)
- Handle ResourceExhausted (rate limit)
- Handle NotFound (missing resources)
- Handle InvalidArgument (bad input)
- Log errors but continue with other resources
- Retry transient errors (3 attempts)
- Skip resources that fail consistently

SECURITY:
- Never log service account keys

- Use Workload Identity when possible
- Validate all input parameters
- Sanitize resource IDs before logging
- Use encrypted secrets management
- Rotate service accounts regularly
- Limit IAM permissions to minimum required
- Use VPC Service Controls when available

PERFORMANCE:
- Collect costs in parallel by project (asyncio.gather)
- Use batch API calls when available
- Cache Cloud Monitoring queries (expensive)
- Limit date range to avoid large BigQuery results
- Use connection pooling
- Implement request throttling per project
- Use BigQuery query cache

GCP-SPECIFIC CONSIDERATIONS:
- Billing export to BigQuery is required (setup guide in docs)
- Costs update with 24-48 hour delay
- Preemptible VMs save 60-91% vs regular
- Committed Use Discounts require 1 or 3 year commitment
- Sustained Use Discounts applied automatically
- Cloud Monitoring metrics have 60-second granularity
- Multi-project collection requires organization-level permissions

FILE STRUCTURE:
services/cost-agent/
```
├── src/
│   ├── collectors/
│   │   └── gcp/
│   │       ├── __init__.py
│   │       ├── base.py              # Base GCP collector
│   │       ├── billing_client.py    # Billing API client
│   │       ├── bigquery_helper.py   # BigQuery cost queries
│   │       ├── compute_engine.py    # GCE collector
│   │       ├── cloud_sql.py         # Cloud SQL collector
│   │       ├── cloud_functions.py   # Cloud Functions collector
│   │       └── cloud_storage.py     # GCS collector
│   ├── analyzers/
│   │   └── gcp_analyzer.py          # Cost analyzer
```

```
│   ├── storage/
│   │   └── gcp_metrics.py          # ClickHouse storage
│   ├── api/
│   │   └── gcp_costs.py            # API endpoints
│   ├── models/
│   │   └── gcp_models.py           # Pydantic models
│   ├── config.py                   # Updated config
│   └── metrics.py                  # Updated metrics
├── tests/
│   ├── collectors/
│   │   └── test_gcp.py
│   ├── analyzers/
│   │   └── test_gcp_analyzer.py
│   ├── storage/
│   │   └── test_gcp_storage.py
│   ├── api/
│   │   └── test_gcp_api.py
│   └── integration/
│       └── test_gcp_integration.py
├── docs/
│   ├── gcp-collector.md
│   └── gcp-bigquery-setup.md
└── requirements.txt                # Updated with google-cloud-*
```

VALIDATION:
After implementation:
1. Run tests: pytest tests/collectors/test_gcp.py -v
2. Test API: curl -X POST http://localhost:8001/api/v1/gcp/collect
3. Check metrics: curl http://localhost:8001/metrics | grep gcp_
4. Verify ClickHouse: SELECT * FROM cost_metrics WHERE provider = 'gcp'
5. Test analysis: curl http://localhost:8001/api/v1/gcp/analysis
6. Verify Grafana: Open Cost Agent dashboard, check GCP panels
7. Test comparison: curl http://localhost:8001/api/v1/gcp/compare-aws

Generate complete, production-ready code with:
- All collector classes following AWS pattern
- Complete API endpoints with Pydantic validation
- Comprehensive tests (unit + integration)
- Prometheus metrics integration
- ClickHouse storage layer
- BigQuery query helpers

- Complete documentation
- GCP IAM role examples
- Multi-cloud comparison logic

✅ Success Criteria

After completing this prompt, verify:

1. GCP Connection

bash

```
# Test GCP credentials
python -c "from google.cloud import bigquery; client = bigquery.Client(); print('Connected:', client.project)"
# Should print project ID
```

2. Cost Collection

bash

```
# Trigger collection
curl -X POST http://localhost:8001/api/v1/gcp/collect
# Expected: {"status": "started", "job_id": "..."}

# Check costs
curl http://localhost:8001/api/v1/gcp/costs?start_date=2025-10-01&end_date=2025-10-31
# Expected: Cost data by service
```

3. Optimization Opportunities

bash

```bash
# Get opportunities
curl http://localhost:8001/api/v1/gcp/opportunities?min_savings=1000
# Expected: List of optimization opportunities
```

4. Multi-Cloud Comparison

```bash
# Compare AWS vs GCP
curl http://localhost:8001/api/v1/gcp/compare-aws
# Expected: Side-by-side comparison
```

5. Metrics

```bash
# Check GCP metrics
curl http://localhost:8001/metrics | grep gcp_total_monthly_cost_usd
# Expected: gcp_total_monthly_cost_usd{service="Compute Engine"} 65000.0
```

6. Tests

```bash
# Run all GCP tests
pytest tests/collectors/test_gcp.py -v
# Expected: All tests pass

# Check coverage
pytest tests/collectors/test_gcp.py --cov=src.collectors.gcp --cov-report=term-missing
# Expected: Coverage ≥80%
```

🔗 Integration Points

The GCP Cost Collector integrates with:

1. Cloud Billing API

- Retrieves cost and usage data via BigQuery export
- Gets billing account details
- Accesses CUD/SUD information

2. Cloud Monitoring API

- Collects CPU utilization metrics
- Monitors network traffic
- Tracks disk I/O and memory

3. Compute Engine / Cloud SQL / Cloud Functions APIs

- Describes instances/databases/functions
- Gets resource labels
- Retrieves configuration details

4. BigQuery

- Queries billing export table
- Aggregates costs by multiple dimensions
- Performs cost trend analysis

5. ClickHouse (0.3)

- Stores time-series cost data
- Stores resource metrics
- Enables multi-cloud queries

6. AWS Collector (1.2)

- Shares base patterns
- Enables cost comparison
- Provides unified interface

📊 Key Metrics Collected

Metric Category	Examples	Frequency
Cost Data	Total cost, cost by service, cost by project	Daily
GCE Metrics	Instance costs, utilization, idle detection	Hourly
Cloud SQL Metrics	Database costs, connections, storage	Hourly
Cloud Functions	Function costs, invocations, duration	Hourly
GCS Metrics	Bucket costs, storage class distribution	Daily
Opportunities	Preemptible eligible, rightsizing, idle	Daily

🎯 Expected Outcomes

After implementation:

- ✅ **Collect** costs from 5+ GCP services
- ✅ **Identify** idle resources (estimated: 5-15%)
- ✅ **Find** preemptible opportunities (estimated: 60-80% savings)
- ✅ **Detect** underutilized instances (estimated: 20-30%)
- ✅ **Compare** AWS vs GCP costs
- ✅ **Store** 30 days of historical cost data
- ✅ **Expose** GCP metrics via Prometheus

📁 Files to be Created

After running this prompt, you should have:

- [ ] `src/collectors/gcp/__init__.py`
- [ ] `src/collectors/gcp/base.py` (~180 lines)
- [ ] `src/collectors/gcp/billing_client.py` (~250 lines)
- [ ] `src/collectors/gcp/bigquery_helper.py` (~200 lines)
- [ ] `src/collectors/gcp/compute_engine.py` (~380 lines)
- [ ] `src/collectors/gcp/cloud_sql.py` (~220 lines)
- [ ] `src/collectors/gcp/cloud_functions.py` (~180 lines)
- [ ] `src/collectors/gcp/cloud_storage.py` (~180 lines)
- [ ] `src/analyzers/gcp_analyzer.py` (~320 lines)
- [ ] `src/storage/gcp_metrics.py` (~230 lines)
- [ ] `src/api/gcp_costs.py` (~280 lines)
- [ ] `src/models/gcp_models.py` (~140 lines)
- [ ] `tests/collectors/test_gcp.py` (~480 lines)
- [ ] `tests/integration/test_gcp_integration.py` (~280 lines)
- [ ] `docs/gcp-collector.md` (~120 lines)
- [ ] `docs/gcp-bigquery-setup.md` (~80 lines)
- [ ] Updated `src/config.py`
- [ ] Updated `src/metrics.py`
- [ ] Updated `requirements.txt`

Total: ~3,300 lines of new code

⏱️ Time Breakdown

Task	Estimated Time
Base collector + Billing client	25 minutes
BigQuery helper	20 minutes
Compute Engine collector	35 minutes
Cloud SQL + Functions + Storage	35 minutes
Cost analyzer	25 minutes
ClickHouse storage	15 minutes
API endpoints	20 minutes
Metrics integration	10 minutes
Unit tests	40 minutes
Integration tests	25 minutes
Documentation	20 minutes
TOTAL	**~4.5 hours**

Actual time with Windsurf: **20 minutes (code gen) + 15 minutes (validation) = 35 minutes**

🔄 Next Steps

After 1.3 is complete, proceed to:

NEXT: PROMPT 1.4 - Azure Cost Collector

- Similar structure to AWS/GCP collectors
- Use azure-mgmt-costmanagement API
- Collect Azure VM, SQL, Functions costs

Document Version: 1.0
Status: 🔄 Ready to Execute
Last Updated: October 21, 2025
Previous: PHASE1-1.2 (AWS Cost Collector)
Next: PHASE1-1.4 (Azure Cost Collector)